FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Optimizing Java Code for Mobile Computing: The Android Example

Rui Miguel Rodrigo Freixedelo

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Manuel Paiva Cardoso (PhD)

July 17th, 2014

Optimizing Java Code for Mobile Computing: The Android Example

Rui Miguel Rodrigo Freixedelo

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. João Carlos Pascoal de Faria (PhD)

External Examiner: Prof. José Gabriel de Figueiredo Coutinho (PhD)

Supervisor: Prof. João Manuel Paiva Cardoso (PhD)

July 17th, 2014

Abstract

Java is one of the most popular programming languages and it is being used in a comprehensive manner across multiple systems due to its versatility. Being a versatile language, it allows more potential for code optimizations.

As embedded and mobile computing devices are limited in terms of computational resources, it is important to optimize applications to be executed on these devices. There are several optimizations of Java code that can be manually applied by developers, but their automatic application is seen as an important task for supporting developers. In this work we consider the code optimizations regarding code motion of certain operations in loop conditions, the substitution of the use of the class String by the class StringBuffer and the initialization of objects.

We developed a prototype compiler, based on the open source tool Spoon, which analyzes and transforms Java code. The compiler is able to analyze the code and to find possible code locations where the optimizations can be applied. The compiler also supports the use of directives by developers to instruct the application of those optimizations.

The compiler was tested using a number of Java applications and the results show some performance gains.

Keywords: Java, Android, Optimization techniques, Performance, Code transformations.

Resumo

Java é umas das linguagens de programação mais populares e tem vindo a ser usada de uma forma abrangente em várias plataformas devido à sua versatilidade. Sendo uma linguagem versátil, existe um maior potencial para otimizações no código.

Tanto os sistemas embebidos como os dispositivos móveis de computação são limitados em termos de recursos computacionais, sendo importante otimizar aplicações executadas nesses dispositivos. Existem várias otimizações de código Java que podem ser aplicadas manualmente pelos programadores, mas a sua aplicação automática é vista como uma tarefa importante no apoio aos programadores. Neste trabalho consideraram-se as otimizações de código sobre o movimento de código de certas operações nas condições de Loop, a substituição do uso da classe String pela classe StringBuffer e na inicialização de objectos.

Foi desenvolvido um protótipo de compilador, com base na ferramenta de código livre Spoon, que analisa e transforma código Java. O compilador é capaz de analisar o código fonte e encontrar possíveis locais onde as otimizações de código podem ser aplicadas. Para além disto, também suporta a utilização de diretivas por parte dos programadores para instruir o compilador dessas otimizações.

O compilador foi testado usando uma série de aplicações Java, e os resultados mostram alguns ganhos de desempenho.

Palavras-chave: Java, Android, Técnicas de otimização, Performance, Transformações de código.

Acknowledgments

A big thank to my supervisor Professor João Cardoso for all the guidance, help, advices tips, and mostly, for all the patient. Without is valuable leading this work would not be the same.

A thank to all my fellows on SPeCS for all help and for receiving me so well.

A thank to my parents, for everything.

A thank to my brother and his family for everything they have done for me.

A thank to my grandparents for all the love.

A thank to Susana for all the love, affection, strength and support.

A thank to my friends, for all the enjoyable times during this work.

Contents

1. Introduction 1
1.1 Context and Motivation 1
1.2 Objectives 3
1.3 Contributions 3
1.4 Organization of the Dissertation 4

2. Related Work 7
2.1 Android Optimizations 7
2.1.1 Bytecode Optimization 7
2.1.2 Computation Offloading 8
2.2 Java Code Optimizations 9
2.2.1 Text Concatenation 9
2.2.2 Code Motion 10
2.2.3 Adequate Object Initialization 10
2.2.4 Object Reuse 11
2.2.5 Object Inlining 11
2.3 Summary 13

3. Frameworks 15
3.1 Frameworks Evaluation 15
3.1.1 FindBugs 16
3.1.2 PMD 16
3.1.3 Checkstyle 17
3.1.4 Spoon 17
3.1.5 JTransformer 17
3.1.6 Soot 17
3.2 Overview 18
3.3 Summary 18

4. Prototype 21
4.1 Spoon 21
4.2 Annotations 23

4.3 Code Transformations 24
4.3.1 Code Motion in Loops 24
4.3.2 String Conversion 26
4.3.3 Object Initialization 26
4.4 Summary 27

5. Experimental Results 29
5.1 Methodology 29
5.2 Static Analysis 30
5.3 Automatic Transformations Results 31
5.3.1 HTML Parser 31
5.3.2 0xbench 33
5.4 String Transformation Results 34
5.5 Summary 34

6. Conclusions 37
6.1 Concluding Remarks 37
6.2 Future Work 38

References 39

List of Figures

Figure 1.1: Worldwide Smartphone OS Market Share (Share in Unit Shipments) (IDC 2014). 2
Figure 1.2: Compiler flow and prototype integration with Spoon (Pawlak, Noguera, and Petitprez 2006). 4
Figure 2.1: Text concatenation (Equivalence between String and StringBuffer). 9
Figure 2.2: Code example of code motion. 10
Figure 2.3: Object initialization example. 11
Figure 2.4: Example of object inlining (Before inlining). 13
Figure 2.5: Example of object inlining (After inlining). 13
Figure 4.1: Code part of the Spoon Java meta-model (partial). Source: (Pawlak, Noguera, and Petitprez 2006). 22
Figure 4.2: Code part of the Spoon Java meta-model (partial). Source: (Pawlak, Noguera, and Petitprez 2006). 23
Figure 4.3: Code motion in loop of type for example. 25
Figure 4.4: Tranformation example with use of annotation. 25
Figure 4.5: StringTransformation annotation specification. 26
Figure 4.6: Initialization annotation specification. 26

List of Tables

Table 1: Object Inlining speedups overview. 12
Table 2: Open Source Java to Java tools. 16
Table 3: Benchmarks main characteristics. 30
Table 4: Benchmarks loops of type for analisys. 30
Table 5: Benchmarks Strings concatenations analysis. 31
Table 6: Methods with more loops transformed by the compiler. 32
Table 7: HTML Parser profiling results. 32
Table 8: Most Representative Hot Spots Methods (original HTML Parser). 32
Table 9: Performance of the methods transformed. 33
Table 10: 0xbench profiling results. 33
Table 11: Most representative methods with loops changes. 34
Table 12: String to StringBuffer transformation speedups. 34

Abbreviations

AST	Abstract Syntax Tree
DBT	Dynamic Binary Translation
DVM	Dalvik Virtual Machine
GUI	Graphical User Interface
ms	Milliseconds
OS	Operating System
OI	Object Inline
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter introduces the dissertation topic, the background context, and the motivation. Then follows a description of the problem and a definition of the main goals.

1.1 Context and Motivation

The Java programming language is an object-oriented programming language, which allows application developers to write a program that runs everywhere on the internet (Gosling et al. 2005) or in different devices (Tyma 1998). This flexibility and simplicity brought popularity to this programming language since the release of version 1.0 in 1996 (Gosling et al. 2005). However, being flexible can lead developers to a non-focused implementation and might make some applications less efficient, which opens opportunities to improve the performance of Java applications.

Android is a Linux based operating system owned by Google in which users can develop a Java based application and deploy it on an Android device (Hall and Anderson 2009). When this operating system (OS) was commercially released in 2008 (Morrill 2008), the number of applications developed to run on this OS was started to growing fast also due to the fact that the applications are developed in Java, a popular and well known programming language (Hall and Anderson 2009). Other support for Android applications development are the Google developed libraries (Grønli, Hansen, and Ghinea 2010). Nowadays, Android is the most sold mobile OS in the world, having reached 81% of worldwide market share in the first quarter of 2014 (IDC 2014).

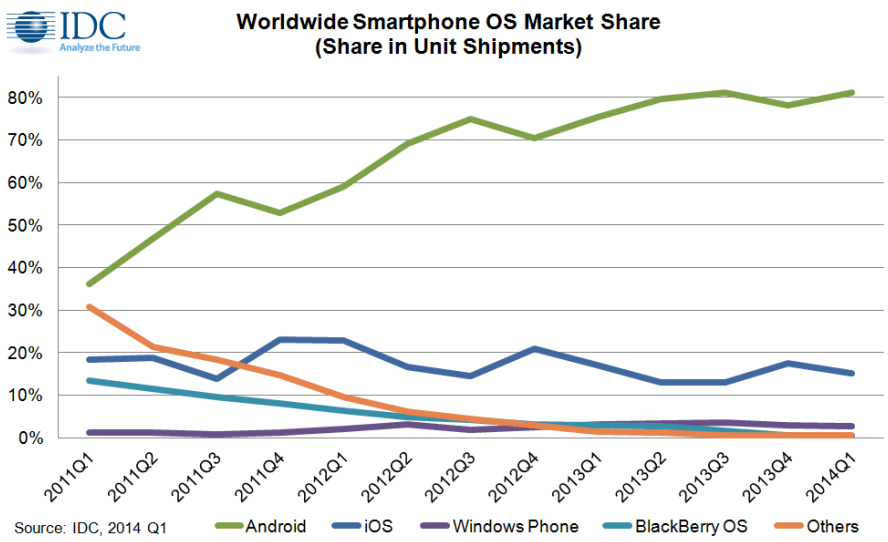

Figure 1.1: Worldwide Smartphone OS Market Share (Share in Unit Shipments) (IDC 2014).

There are many optimization techniques for Java code, which are mainly referred as programming tips and which can improve applications performance. Those optimizations that can be applied to Android Java applications are the aim of this dissertation.

As in all mobile devices, the ones that run Android OS have a set of constraints mainly related to computation capability. Common users want more computational capacity but without the loss of battery capacity. This happens because portability is a major characteristic of mobile devices, hence the term “mobile”. One way to take advantage of mobile resources is by optimizing the Java code in order to improve performance.

The process involved to develop software for mobile devices is similar to the one used to develop other software. Generally four main phases are taken in account: design, implementation, testing and release. Performance improvements are important in order to release the application to the market.

Java code optimization for Android applications is important due to a number of reasons. One of those reasons is that many users have old devices with outdated resources and the applications are evolving daily and using more and more computational resources. Even the new smartphones are sold with a large range of specifications. Other reason is that the most recent devices or the devices with better resources have benefits with applications with better overall performance, that lead to lower energy consumption which is of major importance to a mobile device user in terms of the lifetime capability of their device. Furthermore, there are also many optimizations that can be applied to a Java application leading to a better overall performance.

1.2 Objectives

This dissertation has two main objectives. The first one is to compile a list of Java code optimization techniques and evaluate their behavior in an Android context. The second one is to build a compiler prototype that automatically detects and implements some of those Java code transformations/optimizations.

For the first goal a set of Java code optimizations are identified and evaluated in order to determine which ones have the potential to improve the performance of Android mobile applications. The second goal involves the development of a framework prototype that detects where these optimizations can be performed, alerting the programmer and in a later phase, with programmer directives passed as annotations, automatically implements a group of selected optimizations to an Android application.

1.3 Contributions

This dissertation contributes to the current state of the art by providing a compiled list of Java code optimization techniques and its potential performance gains as well as a prototype that applies some of those optimizations.

The prototype has been developed using Spoon (Spoon 2014) extending some of the already implemented functions. Figure 1.2 shows the compilation flow. The input Java code can be extended with annotations and is analyzed by the Spoon core function, fulfilling the Spoon model according to which type of code part. The transformations are applied according to the given instructions of the programmer and the final output is the transformed Java code.

This prototype helps developers writing code with better performance, which shows the relevance and importance of the work presented in this dissertation. The potential of the developed prototype is evaluated in a set of performance tests.

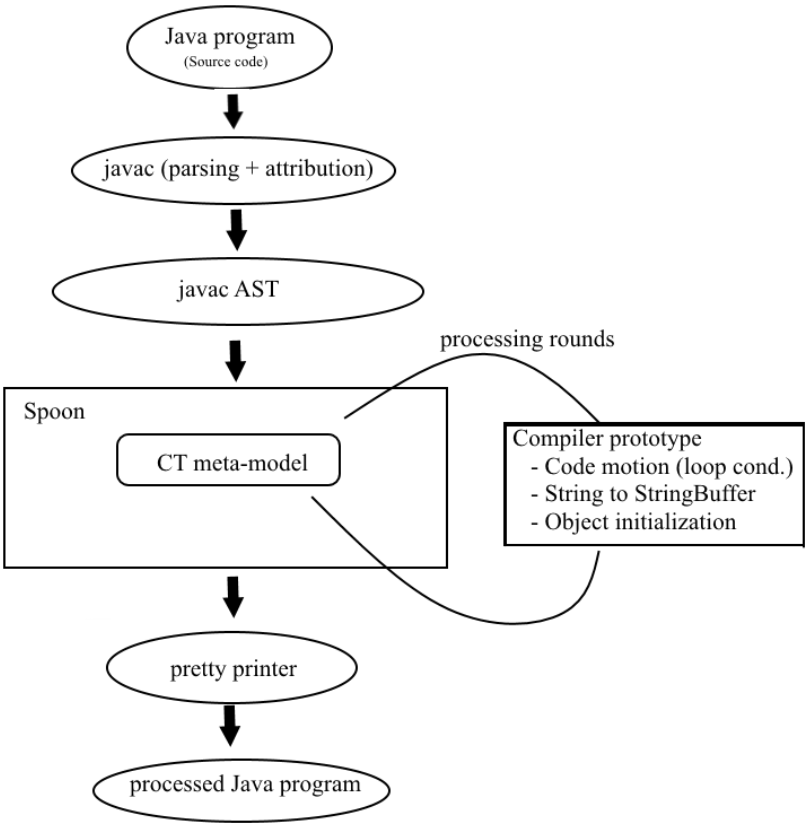

Figure 1.2: **C**ompiler flow and prototype integration with Spoon (Pawlak, Noguera, and Petitprez 2006).

This prototype also contributes to Spoon (Spoon 2014) due to the type of transformations that was applied. This optimizations rewrites the class AbstractManualProcessor which is extended to apply the transformations. We rewrite this class in order to control how the elements are searched in the spoon meta-model. Using the AbstractManualProcessor is possible to visit all elements in Spoon meta-model and then make the necessary analysis and apply all the transformations. The process of visit each element in Spoon meta-model is called processing rounds.

The compiler works based on developer directives, written as annotations. These directives are necessary due to some analysis that cannot be automatically performed. With the use of the developed compiler, some performance improvements are expected.

1.4 Organization of the Dissertation

The rest of this document is organized into five chapters. The second chapter consists in the presentation of related work, namely code transformation techniques. The third chapter focuses on the choice of the open source framework used as a basis to develop the prototype.

The fourth chapter explains the work developed, with the description of the strategies used to build the prototype. The fifth chapter shows the experimental results. The last chapter includes the dissertation main conclusions and provides possible directions for future work.

.

Chapter 2

Related Work

This chapter contains an overview of approaches related with the main focus of this work. The described approaches are related to optimization techniques applied to Java or Android applications. The chapter is divided in two main sections. The first section encompasses optimizations techniques that optimize the Java applications without recurring to code modifications. The second section focuses on Java code optimizations and how these optimizations can improve performance in specific contexts.

2.1 Android Optimizations

There are various forms to optimize a Java application. In this section we analyze strategies that exist to optimize Java or Android applications. Despite the performance gains, some optimization techniques may lead to loss of portability and flexibility (Kazi et al. 2000). Usually, the main goal of applying optimizations is to achieve a better performance with an application using the same resources.

2.1.1 Bytecode Optimization

The compilation of Java code generates the bytecodes that Java virtual machines execute. The bytecodes generated are platform-independent which makes Java programs portable across several hardware platforms. Because the generated bytecodes have no knowledge of the target on which the code will be executed, some translation or interpretation must occur at run-time, which adds to the application's execution time (Kazi et al. 2000). Reordering or changing bytecodes instructions can make methods faster. Usually, this is achieved by reducing the number of instructions, or sometimes by increasing the number of instructions (Shirazi 2000).

AccelDroid (Wang, Wu, and Cintra 2013) is a tool which accelerates the Android Dalvik bytecode execution on the Hardware/Software co-designed processor through direct bytecode translation in the dynamic binary translation (DBT). In Android Java Bytecodes are executed through the Dalvik Virtual Machine (DVM) that translates to Instruction Set Architecture (ISA). The proposed approach eliminates this inefficiency by translating code to ISA only once. AccelDroid was tested on an X86 based Android system with version 2.3, and the results show performance improvements by 78% and energy savings by 40% for the CaffeineMark 3.0 (CaffeineMark 1997) benchmark (Wang, Wu, and Cintra 2013).

2.1.2 Computation Offloading

The nowadays facilitated Internet access achieved by technological evolutions at Internet level ends in this idea: a mobile device is always connected. Based on that, offload of applications has been proposed, for optimization of power and performance on Android mobile devices. This consists on running tasks of an application on the cloud instead of on the own device.

Computation offloading is one of the promising solutions to improve performance of mobile applications in general and Android applications in particular, leading to device's energy consumption gains. However it is necessary a detailed evaluation to apply offloading for many reasons. One of this reasons is that some code can run only on the smartphone, for example to use device sensors; other reason is that the reduction in execution time must be greater than the network delay caused by computation offloading (Zhang et al. 2012).

DPartner (Zhang et al. 2012) is a tool that automatically analyzes the Java bytecode of an application, rewrites it in order that one part of the application runs on the Android platform and the other part runs on the server. This tool automatically divides the classes into two categories: Anchored and Movable. The Anchored classes must run on smartphone because use some device resources, such as GPS, are only available for use if code calls are on device. The Movable classes are transformed using a proxy and are offloaded in packages composed by classes that interact frequently. This package class's aggrupation avoids the time-consuming network communication between these classes, and helps accelerate runtime decision. According to the authors this solution can obtain reductions of execution time by 46% to 97% and battery power consumption by 27% to 83% (Zhang et al. 2012).

AppMobiCloud (Wang et al. 2013) is another tool that works on web based mobile applications based on JavaScript. This tool automatically selects parts of JavaScript code to run on the server side leading to reductions of applications' execution time up to 98% and energy consumption up to 83% on mobile devices (Wang et al. 2013).

Despite the gains that can be obtained with application offloading optimization there is a great problem due to a necessity of a permanent connection to Internet. First not always devices have possibilities to connect to Internet. Second, to apply this optimization is often necessary a

fast Internet connection, which is not always ensured. Third, the Internet connections that are not Wi-Fi often have restrictions to the amount of data that can be transferred leading to constraints in the use of this optimization.

In sum, it is achievable to save energy by offloading parts of an application, which mainly consume device's energy by communicating through Internet instead of local computation. However, the savings depend on several conditions such as, network conditions, traffic patters, and the type of network used (Saarinen et al. 2012). Also server's constraints can be an issue to implement this optimization, for instance if a server is shared by users the occupation can lead to inefficiencies related with server use.

2.2 Java Code Optimizations

Java programs can be optimized recurring to several programming techniques. Although compilers make some optimizations, there are bottlenecks that are currently only be resolved by developers (Shirazi 2000).

2.2.1 Text Concatenation

Generally manipulation of string is done using the `String` class and operations like strings concatenation are done using the '+' operator. This is not efficient and occurs because `String` is immutable and this operations results in a creation of various intermediate objects. The alternative is to use `StringBuffer` with the append operator (Shirazi 2000, Narayanan and Liu 1999).

To increase the performance of repeated string concatenations, it is recommended the use of `StringBuffer` class to reduce the number of intermediate String objects that are created for evaluation of an expression (Gosling et al. 2005).

Figure 2.1 shows a code snippet that illustrates the equivalence described that leads to performance gains.

```
String a = "Hello ";
a += "world!";

//Equivalent to:

StringBuffer a = new StringBuffer("Hello ");
a.append("world!");
```

Figure 2.1: Text concatenation (Equivalence between String and StringBuffer).

2.2.2 Code Motion

Programs spend most of their execution time in a small portion of code (Davidson and Jinturkar 2001). Optimizations able to improve the performance of this small portion will likely lead to an overall improvement in performance. Most of times this small portion of code often corresponds to loops (Villarreal et al. 2002, Kobayashi 1984) and thus loop optimizations are an important step in performance improvements. Loops are one of the code parts in which applications spent more time. For that reason loop optimizations can give significant performance improvements.

There exists some parts of Java code that with a simple change of code place can bring significant performance improvements. For example, calculating the loop stop value outside the loop instead of calculating it in all iterations, can give significant performance improvements (Shirazi 2000). An example of this improvement is shown in Figure 2.2 it consists in moving the stop loop value calculation to outside of the loop. For instance, the compiler do not perform this optimization because cannot know if the array size will not change inside the loop.

```
for (c=0; c<library.size();c++){
      (...)
}

//Equivalent to:

int vecSize = library.size();
for (c=0; c<vecSize; c++){
      (...)
}
```

Figure 2.2: Code example of code motion.

2.2.3 Adequate Object Initialization

Many native Java objects contain buffers that expand dynamically when a need to add more data to them emerge. Some examples of such classes are `ByteArrayOutputStream`, `StringBuffer`, `Vector`, and `Hashtable`. When objects with buffers are created, it is possible to set an initial size. In one hand it is expected to set a size that meets the needs, which means avoiding dynamically increase the buffer size. In other hand allocating a big initial buffer size also decrease performance if the allocated memory is bigger than the needs (Klemm 1999).

In conclusion, if possible to know a priori, in average, the size of the buffer object to be created, a good initial size attribution can lead to performance gains. An example of the described object initialization is shown in Figure 2.3.

```
//Without initial size
Vector<Integer> objvec = new Vector<Integer>();
(...)

//With initial size of 400
Vector<Integer> objvec = new Vector<Integer>(400);
(...)
```

Figure 2.3: Object initialization example.

2.2.4 Object Reuse

Objects are expensive to create and if it is reasonable to reuse the same object, this can give good performance results. Instead of calling `new` to create a new object of the same class, it is more efficient to reset the fields and then reuse the object if the previous information contained in the object is anymore needed (Shirazi 2000).

2.2.5 Object Inlining

Object inlining is one of the source code level Java optimization techniques specific to object-oriented programming languages. However, a complex analysis to perform this code transformation is needed. The object inlining optimization consists in an object from a class that is inlined, being the fields and methods called without recurring to a creation of an object. There have been several authors considering this optimization because the performance gains can be significant (Tyma 1997).

(Ben Asher et al. 2012) proposes a semi-automated technique to object inlining which is divided in two stages. First, an automatic analysis of the source code is made and informs the user, via comments in the IDE, about code transformations that are needed in order to enable or to maximize the potential of the object inline optimization. In the second stage, the object inlining optimization is applied automatically to the source code as a code refactoring operation, or preferably, as part of the compilation process prior to run the Java compiler (e.g., javac) responsible to generate the Java bytecodes.

Table 1 presents examples focusing on object inlining and depicts some of their characteristics and performance impact.

Table 1: Object Inlining speedups overview.

Author	Benchmarks	Target	Speedup
(Ben Asher et al. 2012)	(SPEC JBB2000 2006)	Intel Core 2 Quad Q6600, 2.40 GHz, with an L1 cache 4×32 KBytes, 8-way set associative, and an L2 cache 2×4096 KBytes, 16-way set associative.	46% - Average improvements for operations per second.
(Lhoták and Hendren 2005)	Compress and db.	Intel Pentium II, 333 MHz, 384MB of memory running Linux 2.2.20. Sun Ultra SPARC-III, 750 MHz. 1GB of memory running SunOS 5.8.	10% (Average)
(Cavazos and O'Boyle 2005)	(SPEC JVM98 2008) and (DaCapo 2009) +JBB.	Intel Pentium-4, 2.8 GHz based Red Hat Linux workstation with 500M RAM and a 512KB L1 cache. Apple Macintosh system with two 533 MHz G4 processors, model 7410 with 640M RAM and 64KB L1 cache.	15% (Average) 37% (Maximum) Total execution time

Figure 2.4 and Figure 2.5 shows an example of object inlining. A class “Line” that have a call to a method of class Point that returns the distance between that point to a point given as argument of the method call. After inlining, the class Point disappears, being its methods embedded on class Line.

```
public class Line {
	private  Point initialPoint, finalPoint;

	public double getLength(){
		return initialPoint.length(finalPoint);
	}
	(...)
}

public class Point {
	public int x, y;

	public double length(Point endPoint) {
		return Math.sqrt(Math.pow(endPoint.x-x, 2.0) +
				Math.pow(endPoint.y-y,2.0));
	}
	(...)
}
```

Figure 2.4: Example of object inlining (Before inlining).

```
public class Line {
	private  int initialPointX, initialPointY,
		finalPointX, finalPointY;

	public double getLength(){
		return Math.sqrt(Math.pow(finalPointX-initialPointX, 2.0) +
				Math.pow(finalPointY-initialPointY,2.0));
	}
	(...)
}
```

Figure 2.5: Example of object inlining (After inlining).

2.3 Summary

In this section we studied some of optimizations techniques both directed to Android devices and Java applications. However it is necessary to take in to account the limitations of this kind of optimizations like the computation offloading which is very dependent on Internet speed. We presented code optimization techniques to apply to Java code by programmers. We spotlight the previous five discussed techniques taking into account the literature existent about this and the overall performance that can be achieved by each one of these optimizations.

Chapter 3

Frameworks

This chapter presents the frameworks that were analyzed in order to decide about the framework to be used for code transformations. We analyzed Java to Java frameworks able to analyze Java code and apply transformations. The evaluation performed took into account: date of last update and activity of each development community, programming language used to develop, intermediate representation, mains features and the availability to apply code transformations.

3.1 Frameworks Evaluation

The main goal of the project was to analyze and apply automatic transformations to Java code. To do so, it was decided to start the development using an open source tool that can analyze the Java code and apply the code transformations referred in the previous chapter. In order to choose that tool, a number of available open source tools were analyzed.

Table 2 briefly compares the analyzed tools. We selected six tools to evaluate and we reviewed the main features of these open source tools that analyze Java code. Following is a short description of each tool:

Table 2: Open Source Java to Java tools.

Name	Version	Release date	Programing language	Intermediate representation[1]
FindBugs (FindBugs 2013)	2.0.3	2013-11-22	Java	XML based representation of Java bytecode.
PMD (PMD 2013)	5.1.0	2013-11-02	Java	XML-based representation of AST.
CheckStyle (CheckStyle 2014)	5.7	2014-02-03	Java	AST representation in a GUI.
Spoon (Spoon 2014)	1.6	2013-09-30	Java	AST on personalized meta-model.
JTransformer (JTransformer 2011)	3.0.0	2011-05-09	Prolog	AST on personalized model.
Soot (Soot 2012)	2.5.0	2012-01-22	Java	Four personalized types of bytecodes representation named: Baf, Jimple, Shimple and Grimp.

3.1.1 FindBugs

FindBugs (FindBugs 2013, Balachandran 2013) is an open source static analysis tool that analyzes Java class files looking for programming defects. The analysis engine reports nearly 300 different bug patterns which in turn are grouped into Categories such as Correctness, Bad Practice, and Security. It provides an extensible plugin architecture in which bug detectors can be easily defined, each of which may correspond to several different bug patterns. Despite making code analysis, this tool analyzes java bytecodes and also does not apply code transformations.

3.1.2 PMD

PMD (PMD 2013, Balachandran 2013) is a Java source code analyzer that identifies bugs or potential anomalies including dead code, suboptimal code, overcomplicated expressions and duplicate code. It has an extensive archive of built-in rules that can be used to identify such code. One can specify new rules by writing them in Java and making use of the PMD helper

[1] This intermediate representation has the goal to apply transformations.

classes. Alternatively, one can also compose custom rules that queries the AST of the program. As FindBugs, this tool does not apply code transformations.

3.1.3 Checkstyle

Checkstyle (CheckStyle 2014, Balachandran 2013) is a free open-source static analysis tool for Java, which checks for coding standard violations in source code. The various checks are represented as checkstyle modules, which can be configured to suit a particular coding style. Checkstyle can be extended by writing user-defined checks in Java. As the two previous tools this one also does not apply code transformations.

3.1.4 Spoon

Spoon (Spoon 2014, Pawlak, Noguera, and Petitprez 2006) is a framework for Java program transformations and static analysis developed in Java by INRIA. Spoon is an open and extensible Java compiler, written in pure Java by using Compile-time reflection techniques. This framework allows users to develop *processors* that run through the elements of the AST represented in an own Spoon developed meta-model. This meta-model aggregates the AST in an easy to understand representation which benefits the code analysis. Considering the goals of our project, Spoon can be a suitable option as it have the tools needed such as the ability of applying Java code transformations. Also a small but active community and some literature and documentation about it.

3.1.5 JTransformer

JTransformer (JTransformer 2011) is a Java analyzing and transformation tool based in Prolog. It converts Eclipse projects into logical representations of their ASTs. They call these first-order-logic predicates program element facts (PEFs). With this tool an Eclipse project can be analyzed and manipulated with complex queries and transformations written in Prolog. It is distributed as an Eclipse plug-in.

3.1.6 Soot

Soot (Soot 2012, Vallée-Rai et al. 1999) is a Java optimization framework with some own implemented optimizations in Java bytecodes, which consists of transforming Java Bytecode subsequently to the implemented intermediate representations: Baf, Jimple, Grimp, and then back to Baf, and lastly to bytecode, and while in each representation, performing some appropriate optimization. Despite this framework represents a possible choice for the project,

the fact of evaluating Java bytecodes is a disadvantage as we interested on a source to source tool.

3.2 Overview

Tools were analyzed for a number of characteristics in order to evaluate their capacity and potential. It was intended that the chosen tool resulted from an analysis based on the following criteria: date of last update and activity of each development community, programming language used to develop, intermediate representation, its mains features and the availability to apply code transformations. After analyzing the main characteristics of the referred frameworks the conclusions are as follows.

The first three tools, FindBugs (FindBugs 2013), PMD (PMD 2013) and CheckStyle (CheckStyle 2014), are only analyzers and do not do any kind of transformations (Balachandran 2013) and their use in the context of this project would need extensive programming efforts. JTransformer (JTransformer 2011) uses Prolog as main programing language, and we preferred one programmed in Java. Soot (Soot 2012, Vallée-Rai et al. 1999) analyzes bytecodes instead of Java source files and thus limiting the action of the transformations and the use of annotations.

The tool that is more suitable for our goals is Spoon (Spoon 2014, Pawlak, Noguera, and Petitprez 2006), as all other frameworks analyzed have crossed excluding points and also presented some described features that fit to the project necessities. These features are mainly the possibility of apply transformations recurring to the already implemented function of inserting code snippets. All elements of the intermediate representation can be visited and analyzed by Spoon which have a major importance for our project.

3.3 Summary

This chapter presented the frameworks analyzed. We considered this analysis very important because it would define the following project steps in terms of implementation, and this decision need to be taken with a large possible of elements analyzed.

We can divide the analyzed frameworks into two main groups, the ones that only analyze Java code and the ones that also apply transformations. We only take in consideration the second group because the first one represents a bigger programming effort, diverting resources from the main objective.

Part of the second group are Spoon, JTransformer and Soot. The selected tool was Spoon because of its mains characteristics that include the ability of apply transformations and its intermediate representation which provides an easy way to manipulate all the information in order to apply code transformations. As JTransformer requires the use of Prolog language to

apply transformations, and Soot analyses Java bytecodes instead of the Java source file, this two frameworks do not represent the best choice for the project.

Chapter 4

Prototype

This chapter describes the prototype implementation using Spoon framework. The prototype is able to analyze Java code in order to identify potential application of code optimizations and automatically apply some of these optimizations. Next, we briefly describe the framework Spoon and the approach in the development of the proposed prototype.

4.1 Spoon

Spoon is a Java program transformation and static analysis framwaork in Java developed by INRIA (*Institut National de Recherche en Informatique et en Automatique*). The project was started in 2006 and is still on development with a new version released in March of 2014 (Spoon 2014).

Besides Java code analysis and transformation, Spoon is also an open and extensible Java compiler, written in Java by using Compile-time reflection techniques (Pawlak, Noguera, and Petitprez 2006) and combines this compile-time reflection with a Java template framework for well-typed and intuitive meta programming (Pawlak 2006).

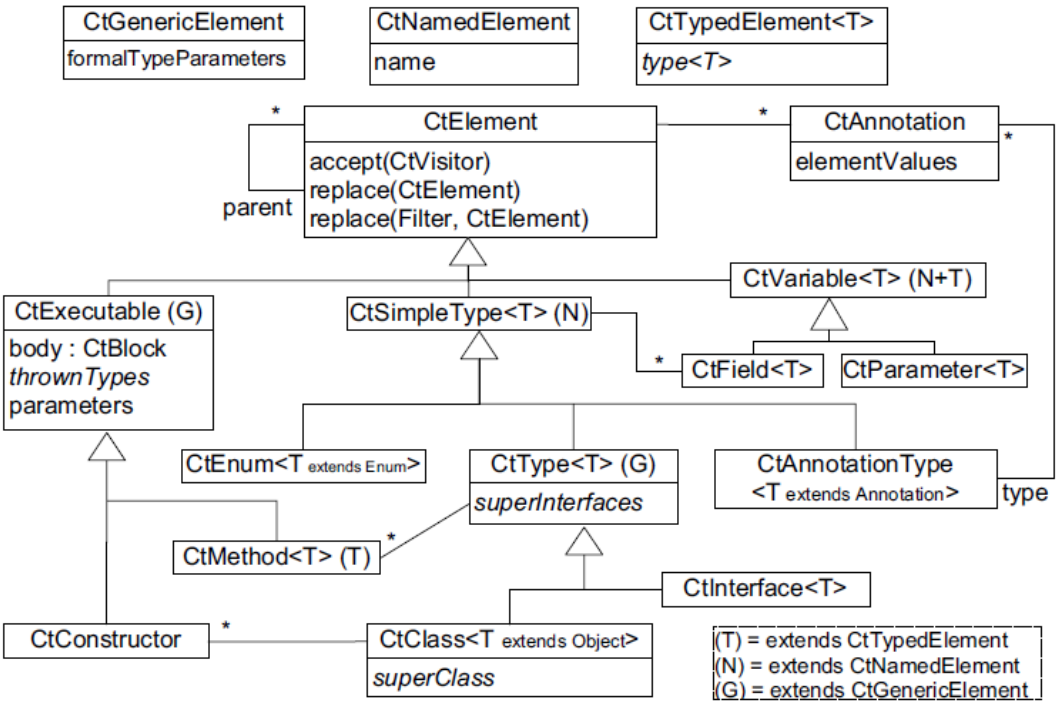

Figure 4.1: Code part of the Spoon Java meta-model (partial). Source: (Pawlak, Noguera, and Petitprez 2006).

To use Spoon potentialities is possible implement a new code processor. At compile-time, Spoon visits the program model and up-call the process method of all the registered processors. The processors can then perform program checks or transformations by using the currently visited element (passed as a parameter) and the compile-time environment, which is set by Spoon, and that allows the processor to access the currently compiled (Pawlak 2005).

Figure 4.2 shows a partial meta-model for the AST of a Java program. There are two main kinds of code elements: the statements (CtStatement), which are un-typed top-level instructions that can be used directly in a block of code, and the expressions (CtExpression), which are typed (type parameter T) and used inside the statements in well-defined contexts. However, some code elements such as invocations and assignments can be used as statements or as expressions depending on the context. In the javac AST, these are enclosed in exec nodes when used as statements. In this meta-model, they simply inherit from both CtStatement and CtExpression (Pawlak, Noguera, and Petitprez 2006).

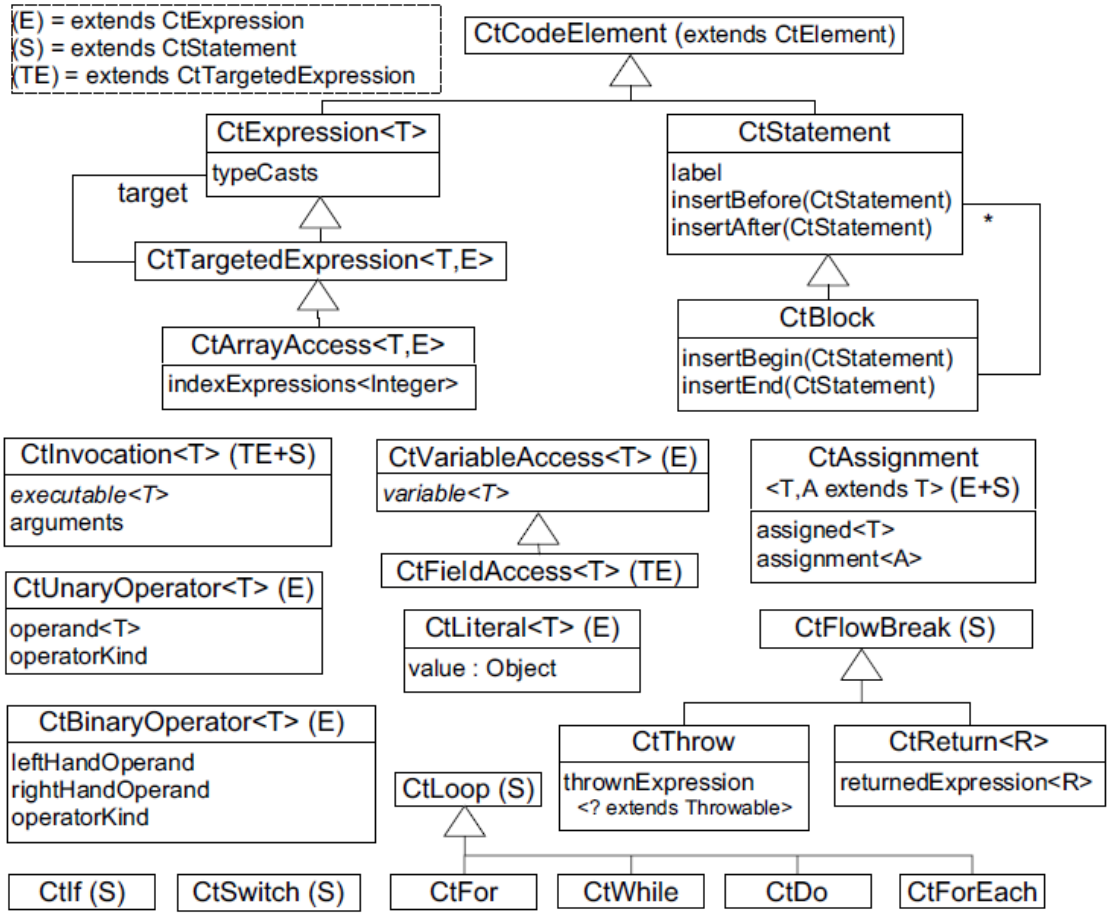

Figure 4.2: Code part of the Spoon Java meta-model (partial). Source: (Pawlak, Noguera, and Petitprez 2006).

Spoon also provides specific types of code elements such as CtLoop. With this element it is possible isolate loops inside a specific part of code and manipulate them. Also, all Spoon elements provide intuitive getters on contained properties and elements simplifying the intended code manipulation.

In sum is possible to create a processor with a specified type of element, passed as processor argument, which will iterate over all elements of that type represented in the AST Spoon meta-model.

We developed three processors in order to fulfill our goal. Each one of these processors are described below.

4.2 Annotations

In order to apply some transformations in some parts of code it is necessary to use programmer directives. This directives can be given by the programmer before running the prototype or after a static analysis.

If the programmer uses our prototype to make a static analysis, is alerted where optimizations can be made, and what are the risks. For instance, if an end loop condition uses an array which is size value is changed inside a loop by a programmer develop method, this means that this optimization cannot be performed. Other example is one `String` that cannot be transformed into `StringBuffer` because in the program some specific `String` methods are used. To deal with this kind of issues, in all implemented transformations, the user is able to set simple annotations in methods that allow our prototype to understand where transformations can be applied.

4.3 Code Transformations

Our approach consists in the use of the Spoon capabilities in order to visit the AST elements of the code. This approach was intended to do a static analysis in order to evaluate the availability of applying some Java code transformations. After performed this analysis, the potential optimization parts of the code are signalized to the programmer. Some of these optimizations are automatically performed.

Before automatically applying transformations, all the transformation phases implemented make a static analysis of the code. This analysis allows making an overall understanding of the potential improvements that can be made. In sum, our prototype is divided in three main parts. The first one is the analysis which outputs a static analysis of the source. The second is the reading and interpretation of the directives given by programmers in methods annotations. And the last one is the program transformation.

The element in the Spoon model that englobes all elements of AST is the CtElement. In our approach we use CtElement to test every element of AST in order to match the particularities of each transformation.

4.3.1 Code Motion in Loops

The goal of this processor is to implement a code motion of the end condition in a loop of type for. This situation is important because for example in on loop that For that is necessary to make an evaluation of each case, and for that reason to develop this processor we considered multiple situations.

Prototype

```
for (int c=0; c<library.size();c++){
      library.elementAt(c);
      (...)
}
(...)
```

```
int vecSize = library.size();

for (int c=0; c< vecSize;c++){
      library.elementAt(c);
      (...)
}
(...)
```

Figure 4.3: Code motion in loop of type for example.

Figure 4.3 shows an example. In that `for`, the instruction that calculates library size will be performed every time the loop runs an iteration. Our optimization switches the instruction for a variable with the size. However this kind of change can be only applied if inside the `for` block the size of the array is not changed.

Our processor works in three distinct steps. First searches for loops of type `for` that have an end condition, in which an array (or vector, or list...) size is called. Then a search is performed in order to find programmer directives, given as annotations in the parent method. If nothing is stated it is necessary to check if this optimization can be applied. If exist an instruction given by the programmer, the code transformation is performed.

If no directives are given, the prototype then searches for native methods applied to that array inside the body of that loop, such as add or remove. This is because to apply this transformation the size of the array must never change. If this these methods are not used in order to automatically apply this transformation, and are not called other program methods, then it is concluded that the transformation can be performed. If methods defined by the programmer, are called, the directive is necessary in order to apply this transformation.

```
@LoopsConstEnd
public void setAttribute
(Vector<String> nodes){

  (...)
  for (int i=0; i<nodes.size();
i++){
     nodes.assert();
     (...)
  }
  (...)
}
```

```
@LoopsConstEnd
public void setAttribute
(java.util.Vector<java.lang.String>
nodes) {

  (...)
  int size_transf = nodes.size();
  for (int i = 0 ; i<(size_transf);
i++) {
     nodes.assert();
     (...)
  }
  (...)
}
```

Figure 4.4: Tranformation example with use of annotation.

4.3.2 String Conversion

Unlike the previous transformation this one is only applied with programmer directives, and never automatically. This occurs because a change of the type String to StringBuffer can cause malfunction of the application, if a transformed String uses some of its unique methods. One solution was to turn all String into StringBuffer before concatenation and then turning again into String. But this solution does not lead to performance gains because the incremental creation of new objects. So, we decided to apply this transformation only when the programmer uses the defined directive.

```
@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface StringTransformation {


}
```

Figure 4.5: StringTransformation annotation specification.

4.3.3 Object Initialization

This transformation aims to find where arrays or vectors are declared without an initial size value. The systematic allocation of memory decreases performance, and if the programmer previously knows the average size of the structures used, one can achieve better performance results with the specification of the structure initial size.

As the previous transformation, this transformation is only applied when programmer directives are given. This avoid create too much larger arrays than that are really required, using too many unnecessary memory. All the non-sized arrays, list or vectors in the object creation the processors will be signalized for the programmer. Then he can manually correct, or give an annotation for some methods, by defining the average size of the arrays initialized inside that method.

```
@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Initialization {
        int averageValue();
}
```

Figure 4.6: Initialization annotation specification.

4.4 Summary

To apply automatic code transformations it is necessary a deep analysis of the program and its methods. Code transformations imply changes, and many times, these changes can affect the correct work of the application.

Being imperative to avoid application malfunctions, the directives can give full control to the programmer to choose where he/she wants to apply the code transformations. For this reason the transformations are mainly applied by programmer directives. However all the potential transformations found in the code are signalized in the static analysis phase.

Chapter 5

Experimental Results

This chapter uses a set of benchmarks to evaluate the approach presented in this dissertation. We present two experiments: the first is a static analysis with the aim of understanding the potential and the embracing of the transformations; the second analysis focuses on automatic code transformations. The selected set of benchmarks covers Java desktop applications and Android applications.

5.1 Methodology

We have used a suite of benchmarks to evaluate our prototype. Table 3 presents the benchmarks used, their main target, and a brief description. A static analysis and automatic code transformations are performed to the benchmarks. We also collect performance results in order to evaluate the potential gains these optimizations can achieve.

All the experiments were performed on two machines depending on the benchmark running target. The Java benchmarks performance tests were performed using an HP dv6-2170ep with Windows 8.1. This machine has a 2.4 GHz Intel Core i5-520M Processor and 4 GB of RAM DDR3 with an NVIDIA GeForce GT 320M GPU (Graphics Processing Unit). The Java VisualVM (VisualVM 2014) profiling tool was used to profile the benchmarks used.

The Android benchmarks performance tests were done using a Motorola Moto G with Android 4.4.2 KitKat. This phone has a 1.2 GHz Quad Core with a Qualcomm MSM8226 Snapdragon 400 chipset and 1 GB of RAM. The device also has a dedicated GPU, the Adreno 305, and 16 GB of storage. These characteristics place this cellphone in the category of medium to high end in terms of the specs currently available on Android based smartphones. DDMS was used to profile the benchmarks used with the Android target.

The execution times were measured by running the benchmarks twenty one times, discarding the worst run, and averaging the remaining twenty.

Table 3: Benchmarks main characteristics.

Name	Target	Description
HTML Parser (HTML Parser 2006)	Java desktop application	Java library used to parse HTML code.
0xbench (0xbench 2011)	Android application	Android benchmark that performs a series of small program test in Android.
lp-doc-cluster (lp-doc-cluster 2014)	Java desktop application	Document clustering engine
HIPR2 (HIPR2 2004)	Java desktop application	Image processing

5.2 Static Analysis

We started by performing a static analysis in order to evaluate the potential of the benchmarks. This analysis shows the number of code transformations that can be applied.

Table 4 shows the number of loops of type `for` and the number of those loops that are transformed for each of the benchmarks being tested. The transformation referred here is the one presented in Section 4.3.1, which is responsible to move the invariant code related to the size of the data structure (e.g., Vector) from the condition of the loop to before the loop. These results were obtained by compiling and running the resultant code after applying our compiler.

The results show that there is opportunity for the code transformation for 70 (out of 218) for loops for the HTML Parser benchmark which represents 32% of the total loops. Also, for the 0xbench benchmark the number of loops that can be transformed is 7 (out of 209) which represents 3%. For the other two analyzed benchmarks, lp-doc-cluster and HIPR2, none loop shows potential to be transformed.

Table 4: Benchmarks loops of type for analisys.

Name	Number of loops of type `for`	Transformed loops	% transformed loops
HTML Parser	218	70	32.1
0xbench	209	7	3.3
lp-doc-cluster	9	0	0
HIPR2	519	0	0

Running the compiler in order to analyze the amount of String concatenations present in the benchmarks code the results are as follows in Table 5. For the HTML Parser benchmark, the number of String concatenations is 487, for 0xbench benchmark the number of String concatenations present in the code is 803, for Cluster benchmark is 21, and for HIPR2 is 4.

Table 5: Benchmarks Strings concatenations analysis.

Name	String concatenations
HTML Parser	487
0xbench	803
Cluster	21
HIPR2	4

These results show potential in both of the optimizations. As explained in Chapter 4, the String concatenation optimizations are only applied when the developer instructs the compiler for doing so by giving directives using annotations in parent methods.

5.3 Automatic Transformations Results

After applying the automatic transformation to loops of type *for* in which the end loop condition calls a method to return the size of an array or similar, we performed a number of measurements in order to evaluate the resultant code. The following subsections present the results for each of the benchmarks being evaluated.

5.3.1 HTML Parser

The results of profiling the HTML Parser (HTML Parser 2006) application were obtained by parsing the website (HTML - Living Standard 2014). In this benchmark the loop optimization transformed 70 loops distributed by 30 methods, being the most representative shown in Table 6.

However, only two of these methods were called by the benchmark when parsing this particular website. Nevertheless, these two loop transformations bring the performance benefits presented in Table 7. We show the average of the results performed in milliseconds, and also the time of the maximum and minimum execution. The transformations achieved an overall performance improvement of 1%.

Table 6: Methods with more loops transformed by the compiler.

Method name	Number of loops transformed
org.htmlparser.util.ParserUtils.splitTags()	4
org.htmlparser.nodes.TagNode.setAttribute()	3
org.htmlparser.util.ParserUtils.splitButChars()	3
org.htmlparser.util.ParserUtils.splitButDigits()	3
org.htmlparser.util.ParserUtils.splitChars()	3
org.htmlparser.util.ParserUtils.splitSpaces()	3

Table 7: HTML Parser profiling results.

Execution type	**Average**	**Minimum**	**Maximum**
Exec. Time without transformations (ms)	16,240.80	14,601.47	18,802.75
Exec. Time with transformations (ms)	15,959.88	14,398.28	17,497.49
Performance improvements (%)	1	1	7

Table 8 shows the profiling results for the methods that contributed more to the global execution time before applying the transformations. The native method `java.io.PrintStream.println()` consumes around 53% of the global execution time and more than 75% of CPU execution time was spent in Java native functions.

Table 8: Most Representative Hot Spots Methods (original HTML Parser).

Hot Spots - Method	**Average time (ms)**	**Of Average total time**
java.io.PrintStream.println()	9.501,92	53.81%
java.util.zip.GZIPInputStream.read()	1.609,91	9,78%
java.net.HttpURLConnection.getResponseCode()	1.484,74	9,02%
java.lang.StringBuffer.append()	398,42	2,42%
org.htmlparser.tags.CompositeTag.toString()	319,75	1,94%
org.htmlparser.util.sort.Sort.bsearch()	264,78	1,61%
java.lang.StringBuffer.setLength()	258,62	1,57%
java.lang.System.arraycopy[native]()	216,93	1,32%
org.htmlparser.nodes.TextNode.toString()	211,38	1,28%
org.htmlparser.nodes.TagNode.toHtml()	206,97	1,26%
sun.net.www.protocol.http.HttpURLConnection.connect()	192,28	1,17%
org.htmlparser.util.NodeList.newNodeArrayFor()	169,04	1,03%

To obtain the execution time improvements achieved by transforming the two loops, and to understand the real impact of these transformations, we measured the execution time of methods and/or code regions using calls to `System.nanoTime()`. The results present a speedup

between 1.13 and 1.19 over the original time elapsed (see Table 9). This shows that the code transformations contribute to significant performance gains.

Table 9: Performance of the methods transformed.

Method	Number of calls	Time without transformations (ms)	Time with transformations (ms)	Speedup
org.htmlparser.nodes. TextNode.toString()	210,817	1,068.90	900.47	1.19
org.htmlparser.nodes. RemarkNode.toString	3,356	42.85	38.01	1.13

5.3.2 0xbench

The 0xbench benchmark is an Android application and it aims at evaluating several mobile phone performance capabilities using some graphics and mathematical functions. The results of profiling this application were taken by running this benchmark with the following application functions: Math, 2D, 3D, VM.

Table 10 shows the results achieved. In this benchmark our compiler transformed 7 loops and we achieved a speedup of 1.20 in the overall application execution time.

Table 10: 0xbench profiling results.

Execution type	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
No transformations	3,463.81	3,370.71	3,560.48
With transformations	2,883.68	2,256.98	3,392.95
Speed Up	1.20	1.49	1.05

The loops transformed are distributed by the methods shown in Table 11. They are distributed over two classes and 6 methods.

Table 11: Most representative methods with loops changes.

Method name	Number of loops transformed
org.zeroxlab.zeroxbenchmark.Benchmark.getJSONResult()	2
org.zeroxlab.zeroxbenchmark.Benchmark.onActivityResult()	1
org.zeroxlab.zeroxbenchmark.Benchmark.RunCase()	1
org.zeroxlab.zeroxbenchmark.Benchmark.getXMLResult()	1
org.zeroxlab.zeroxbenchmark.Benchmark.getResult()	1
org.zeroxlab.zeroxbenchmark.CaseScimark2.getScenarios()	1

5.4 String Transformation Results

Recurring to the developed annotations we analyzed the source code of the 0xbench benchmark in order to perform transformations in `String` concatenation. We transformed a total of 10 methods in this benchmark resulting in 18 objects of type `String` transformed into type `StringBuffer`. This transformation resulted in a speedup minimum of 1.02 and a maximum of 1.11 as show in Table 12.

Table 12: String to StringBuffer transformation speedups.

Number of changes (String to StringBuffer)	Speed up		
	Average	Minimum	Maximum
18	1.09	1.11	1.02

5.5 Summary

We analyzed a set of benchmarks in order to evaluate the potential of Java code optimizations. The benchmarks analyzed present characteristics that allow the application of some Java code optimization techniques. Commonly Java applications have potential to perform code optimizations. These optimizations need, in most cases, to be applied by the programmer, or led by manually inserted directives. In this chapter we showed that the selected optimizations techniques are present in the analyzed benchmarks. The results showed that the loop transformations lead to performance improvements.

Experimental Results

Chapter 6

Conclusions

6.1 Concluding Remarks

In this dissertation we have studied Java code transformation techniques in order to identify possible sources of performance gains. From the exiting identified code transformations we selected a set of those transformations to automatically apply to Java code. After a carefully analysis of those optimizations, we concluded that, in most cases, to apply the transformations, programmer directives are needed.

We presented a compiler prototype based on the Spoon framework, which simplifies the task of analyses the Java code in order to apply the Java code optimization techniques. The prototype mainly uses directives to apply the transformations.

The contributions of this dissertation are the implementation and the experimental validation of the prototype. By using the prototype we were able to identify possible optimizations, to apply those transformations and to achieve performance improvements.

We conclude that the implemented code optimization techniques achive performance gain in Java applications. Thus, they can support programmers on deciding and applying especially as some of those optimizations are complex and need a deep analysis of the implemented code, even more when we are dealing with an application with many source files and classes.

The developed prototype focuses on the following three types of optimizations:

1. String concatenation, which consists in analyzing where can be applied a change from the variables of type `String` to `StringBuffer`, and applying also a transformation in Strings concatenation by using the append operator of `StringBuffer`. This optimization is performed when the programmer uses directives to instruct the compiler.

2. Motion of loop end condition of type `for`, which consists in analyzing and automatically applying this transformation based on an analysis of the body of for loop. The transformation can be automatically applied, but programmer directives are used to select the possible locations.
3. Object initialization, uses programmer directives to set an initial size when objects are declared without initial size.

We included these optimizations in the compiler prototype and we evaluated it by applying them automatically to Java applications. The resultant code was then evaluated using the execution time reductions of the transformed code when compared to the original code.

6.2 Future Work

Our prototype could also be added with new features such as the implementation of more optimization techniques, analysis and transformations. One of the future transformations could be object inlining. Those features could bring more value to our compiler prototype and lead to higher performance improvements.

References

0xbench. 2011. Accessed June 25, 2014. https://code.google.com/p/0xbench/.

Balachandran, Vipin. 2013. "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation." Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, pp. 931-940.

Ben Asher, Yosi, Tomer Gal, Gadi Haber, and Marcel Zalmanovici. 2012. "Refactoring techniques for aggressive object inlining in Java applications." *Automated Software Engineering* 19 (1):97-136.

CaffeineMark. 1997. "CaffeineMark." Accessed June 25, 2014. http://www.benchmarkhq.ru/cm30/.

Cavazos, John, and Michael F. P. O'Boyle. 2005. "Automatic Tuning of Inlining Heuristics." Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Seatle, WA, USA.

CheckStyle. 2014. "CheckStyle." Accessed June 25, 2014. http://checkstyle.sourceforge.net/.

DaCapo. 2009. "The DaCapo Benchmarks." Acessed June 25, 2014. http://www.dacapobench.org/.

Davidson, Jack W., and Sanjay Jinturkar. 2001. An Aggressive Approach to Loop Unrolling. Technical Report CS-95-1 3, Department of Computer Science, University of Virginia, Charlottesville.

FindBugs. 2013. "FindBugs." Accessed June 25, 2014. http://findbugs.sourceforge.net/.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2005. *Java Language Specification, The*. 3rd ed: Addison-Wesley Professional.

Grønli, Tor-Morten, Jarle Hansen, and Gheorghita Ghinea. 2010. "Android vs Windows Mobile vs Java ME: a comparative study of mobile development environments." Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments, Samos, Greece, pp. 1-8.

Hall, Sharon P., and Eric Anderson. 2009. "Operating systems for mobile computing." *J. Comput. Small Coll.* 25 (2):64-71.

References

HIPR2. 2004. Accessed June 25, 2014. http://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm.

HTML - Living Standard. 2014. "HTML - Living Standard." Accessed June 25, 2014. http://www.whatwg.org/specs/web-apps/current-work/.

HTML Parser. 2006. "HTML Parser." Accessed June 25, 2014. http://htmlparser.sourceforge.net/.

IDC. 2014. "Smartphone OS Market Share, Q1 2014." Accessed June 25, 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

JTransformer. 2011. "JTransformer." Accessed June 25, 2014. https://sewiki.iai.uni-bonn.de/research/jtransformer/start.

Kazi, Iffat H., Howard H. Chen, Berdenia Stanley, and David J. Lilja. 2000. "Techniques for obtaining high performance in Java programs." *ACM Comput. Surv.* 32 (3):213-240.

Klemm, Reinhard. 1999. "Practical guidelines for boosting Java server performance." Proceedings of the ACM 1999 conference on Java Grande, San Francisco, California, USA, pp. 25-34.

Kobayashi, M. 1984. "Dynamic Characteristics of Loops." *IEEE Trans. Comput.* 33 (2):125-132.

Lhoták, Ondřej, and Laurie Hendren. 2005. "Run-time evaluation of opportunities for object inlining in Java." *Concurrency and Computation: Practice and Experience* 17 (5-6):515-537.

lp-doc-cluster. 2014. "lp-doc-cluster." Accessed June 25, 2014. https://code.google.com/p/lp-doc-cluster/.

Morrill, Dan. 2008. "Announcing the Android 1.0 SDK, release 1." Last Modified September 23, 2008 Accessed June 25, 2014. http://android-developers.blogspot.pt/2008/09/announcing-android-10-sdk-release-1.html.

Narayanan, S.N., and J. Liu. 1999. *Enterprise Java Developer's Guide*: McGraw-Hill Osborne Media.

Pawlak, R, C Noguera, and N Petitprez. 2006. Spoon: Program Analysis and Transformation in Java. In *Tech. Rep. 5901, INRIA Research Report*: INRIA.

Pawlak, R. 2006. "Spoon: Compile-time Annotation Processing for Middleware." *Distributed Systems Online, IEEE* 7 (11): p. 1.

Pawlak, Renaud. 2005. "Spoon: annotation-driven program transformation --- the AOP case." Proceedings of the 1st workshop on Aspect oriented middleware development, Grenoble, France.

PMD. 2013. "PMD." Accessed June 25, 2014. http://pmd.sourceforge.net/.

Saarinen, Aki, Matti Siekkinen, Yu Xiao, Jukka K. Nurminen, Matti Kemppainen, and Pan Hui. 2012. "Can offloading save energy for popular apps?" Proceedings of the seventh ACM international workshop on Mobility in the evolving internet architecture, Istanbul, Turkey, pp. 3-10.

References

Shirazi, J. 2000. *Java Performance Tuning*: O'Reilly Vlg. GmbH & Company.

Soot. 2012. "Soot." Accessed June 25, 2014. http://www.sable.mcgill.ca/soot/.

SPEC JBB2000. 2006. "Standard Performance Evaluation Corporation." Accessed June 25, 2014. http://www.spec.org/jbb2000/.

SPEC JVM98. 2008. "SPEC JVM98 Benchmarks." http://www.spec.org/jvm98/.

Spoon. 2014. "Spoon." Accessed June 25, 2014. http://spoon.gforge.inria.fr/.

Tyma, Paul. 1997. "Tuning Java performance." In *Dr. Dobb's Java development tools & techniques*, edited by Kim Eugene Eric, 145-154. Miller Freeman, Inc.

Tyma, Paul. 1998. "Why are we using Java again?" *Commun. ACM* 41 (6):38-42.

Vallée-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. "Soot - a Java bytecode optimization framework." Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, Mississauga, Ontario, Canada, p. 13.

Villarreal, Jason, Roman Lysecky, Susan Cotterell, and Frank Vahid. 2002. A study on the loop behavior of embedded programs. Technical Report UCR-CSE-01-03, University of California, Riverside.

VisualVM, Java. 2014. "Java VisualVM." Accessed June 25, 2014. http://visualvm.java.net/.

Wang, Cheng, Youfeng Wu, and M. Cintra. 2013. "Acceldroid: Co-designed acceleration of Android bytecode." 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 23-27 Feb. 2013, pp. 1-10.

Wang, Xudong, Xuanzhe Liu, Gang Huang, and Yunxin Liu. 2013. "AppMobiCloud: improving mobile web applications by mobile-cloud convergence." Proceedings of the 5th Asia-Pacific Symposium on Internetware, Changsha, China, pp. 1-10.

Zhang, Ying, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. 2012. "Refactoring android Java code for on-demand computation offloading." Proceedings of the ACM international conference on Object oriented programming systems languages and applications, Tucson, Arizona, USA, pp. 233-248.

References